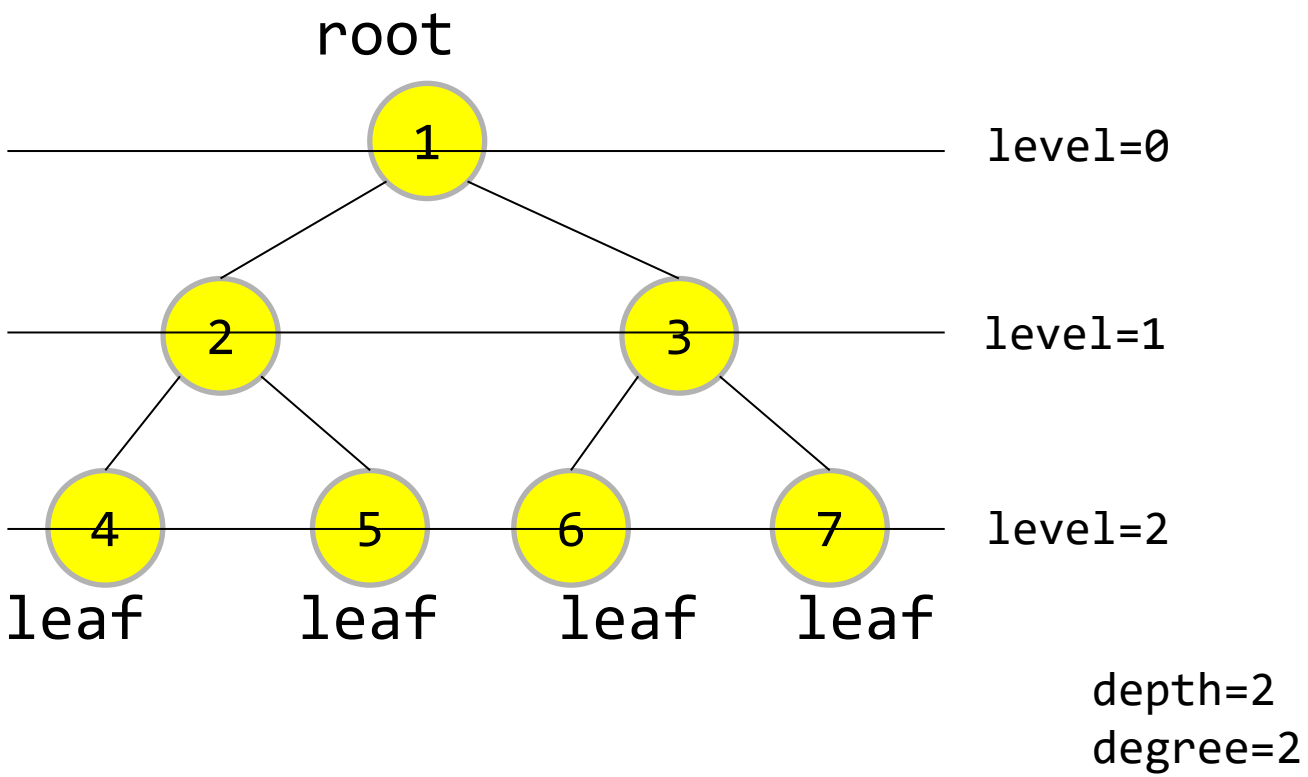
TREE

- 이진 탐색 트리의 구성원리를 이해 한다.
- **RB Tree** 의 밸런스 원리를 이해 한다.

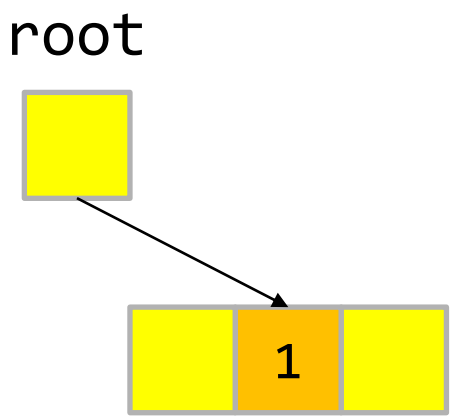
1) 트리의 기본 코드 구현
2) 이진 탐색 트리의 구현
3) RB 트리의 구현 및 분석

Tree 구조


root
1
2
3
4
5
6
7
level=0
level=1
level=2
leaf
leaf
leaf
leaf
depth=2
degree=2

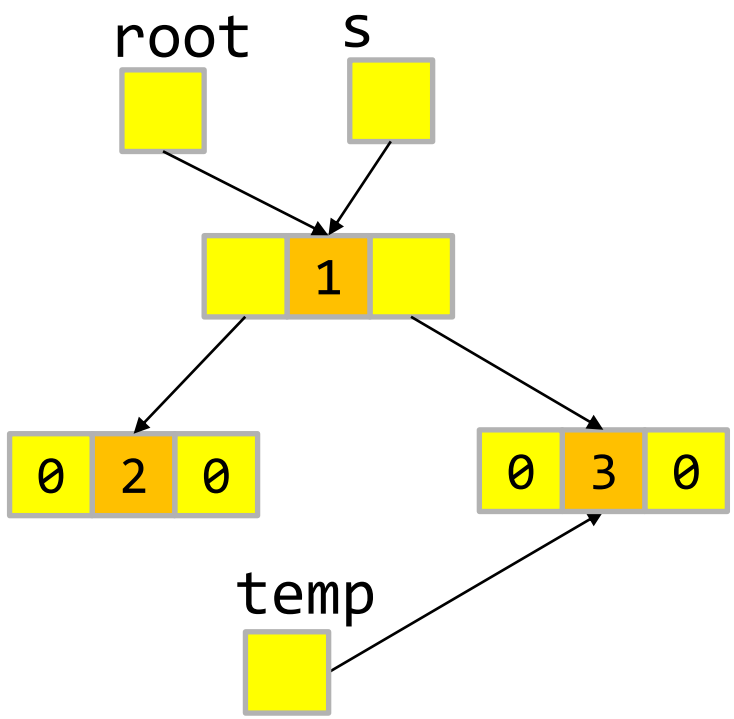
Tree 구조

```
typedef struct _node
{
    int data;
    struct _node *left;
    struct _node *right;
} NODE;

NODE *root;
```



Tree의 insert

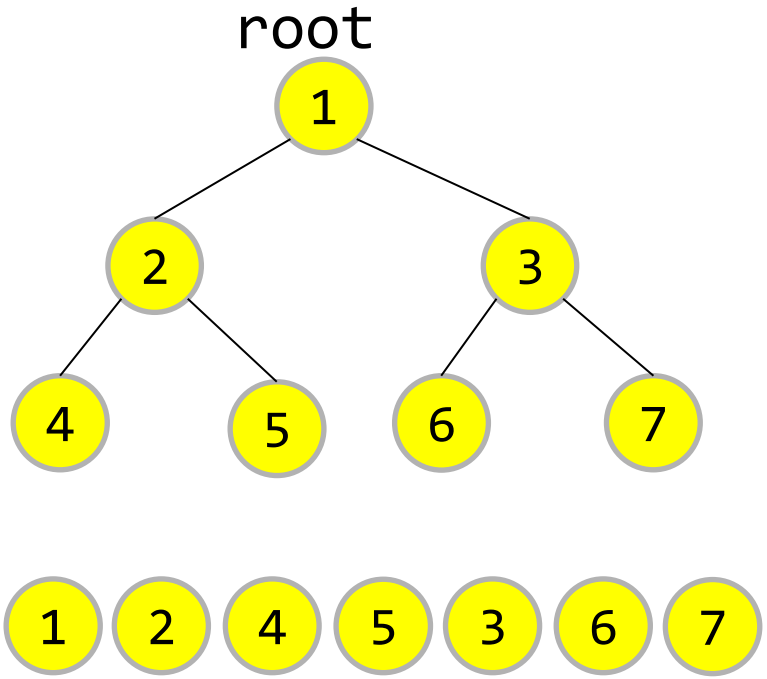
```
typedef enum { LEFT, RIGHT } FLAG;
void insert_data( int data, NODE *s,
                  FLAG flag )
{
   NODE *temp;
   temp = malloc( sizeof(NODE) );
   temp->data = data;
   temp->left = temp->right = 0;

   if( root == 0 )
   {
       root = temp;
       return ;
   }
   if( flag == LEFT )
       s->left = temp;
   else if( flag == RIGHT )
       s->right = temp;
}
```



Tree Order Traverse

트리의 순회(Tree Order Traverse)

pre order : root - left - right

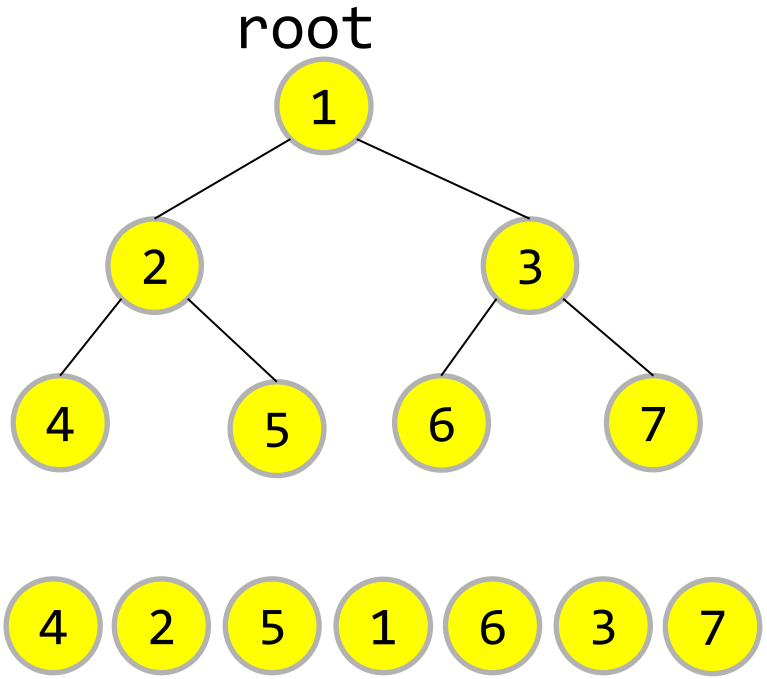



```
void pre_order( NODE *temp )
{
      if( temp==0 )
           return;

      printf("%d\n", temp->data );
      pre_order( temp->left );
      pre_order( temp->right );
}
```

Tree Order Traverse

트리의 순회(Tree Order Traverse)

in order : left - root - right

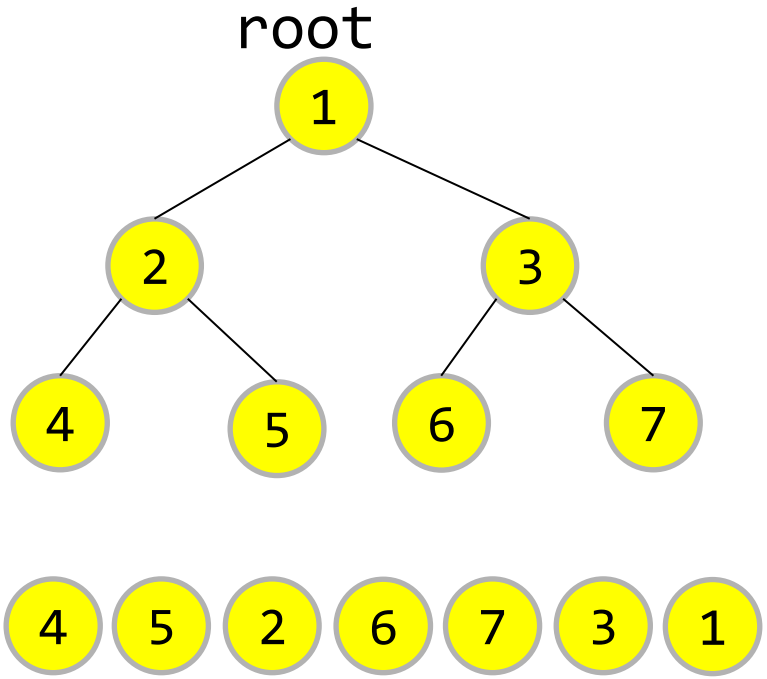



```
void in_order( NODE *temp )
{
      if( temp==0 )
           return;

      in_order( temp->left );
      printf("%d\n", temp->data );
      in_order( temp->right );
}
```

Tree Order Traverse

트리의 순회(Tree Order Traverse)

post order : left - right - root

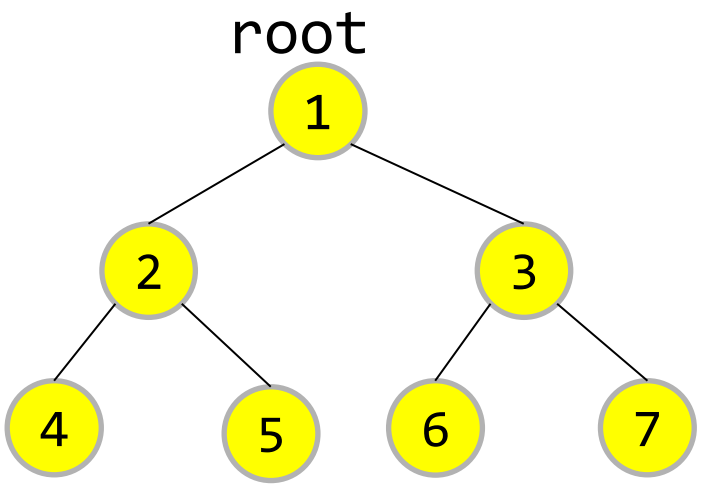



```
void post_order( NODE *temp )
{
      if( temp==0 )
           return;

      post_order( temp->left );
      post_order( temp->right );
      printf("%d\n", temp->data );
}
```

In Order Traverse를 이용한 display 함수의 구현

display : right - root - left

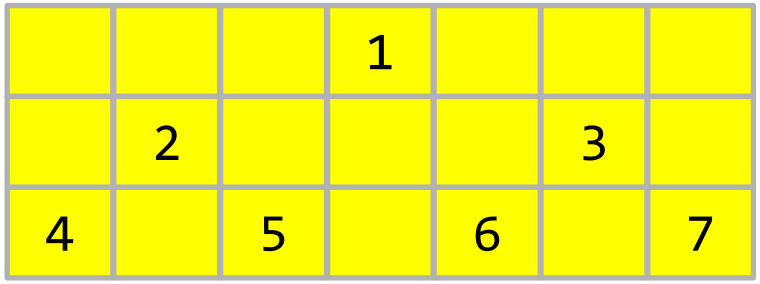
```
void display( NODE *temp )
{
      static int indent=-1;
      int i;

      if( temp==0 )
           return;

      ++indent;
      display( temp->right );
      for(i=0; i<indent; i++ )
              printf("%4c", ' ');
      printf("%d\n", temp->data );
      display( temp->left );
      --indent;
}
```

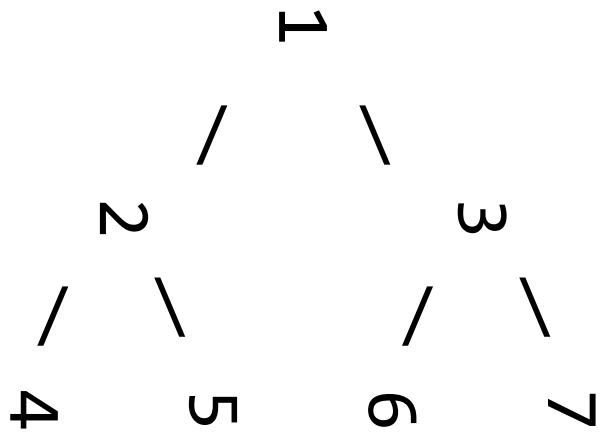


7
3
6
1
5
2
4

배열을 이용한 display 함수의 구현

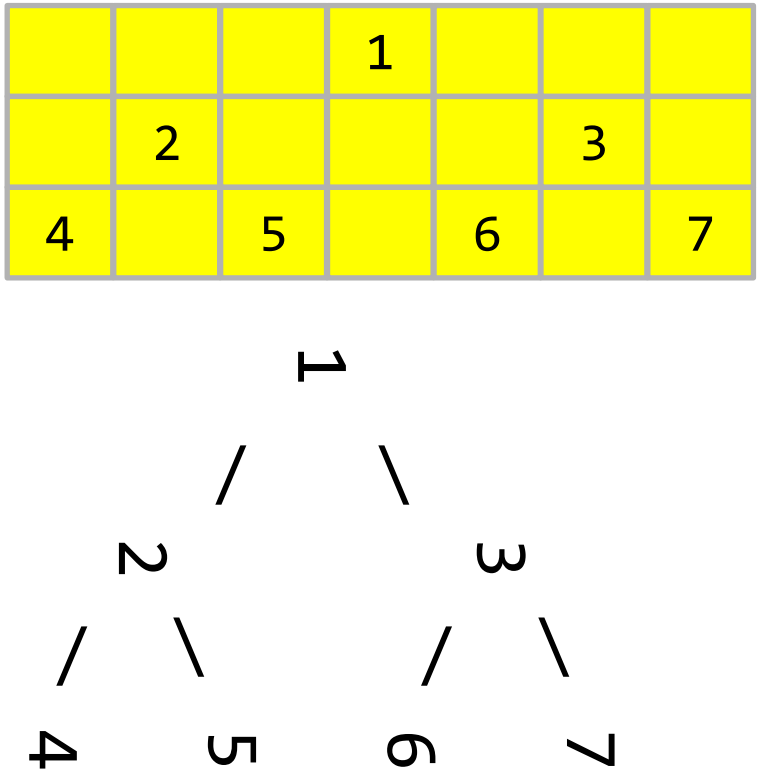
```
void __display( NODE *temp,  )
{
      int i;
      if( temp==0 )
           return a;

      ++row;
      __display( temp->right );
      a[row][col++]=temp->data;
      __display( temp->left );
      --row;
      return a;
}
```





배열을 이용한 display 함수의 구현

```
void display( NODE *temp )
{
      int row=-1;
      int col=0;
      int a[10][10]={0,};
      int i,j;
      __display(temp,a,&row,&col);
      system("clear");
      for(i=0; i<10; i++ )
      {
         for(j=0; j<10; j++ )
         {
             if( a[i][j] )
                 printf("%4d", a[i][j]);
             else
                 printf("%4c", ' ');
         }
         printf("\n");
      }
      getchar();
}
```



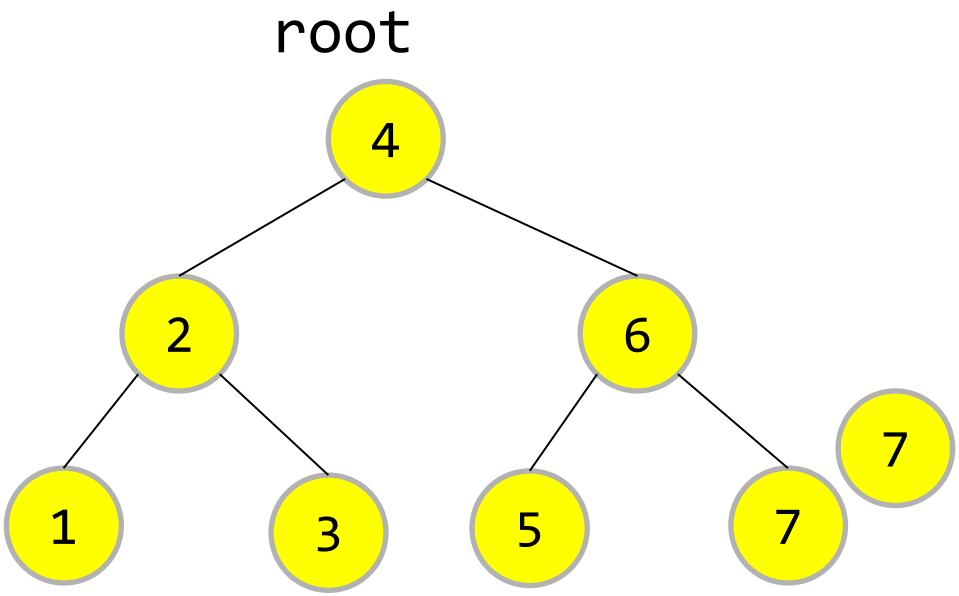
배열과 행과 열을 인자로 처리한 구현 예

```
void __display( NODE *temp, int (*a)[10],
                int *row, int *col )
{
    if( temp == 0 )
        return ;
    ++*row;
    __display( temp->left, a, row, col);
    a[*row][(*col)++] = temp->data;
    __display( temp->right, a, row, col );
    --*row;
}
```

이진 탐색 트리

Binary Search Tree 구조

: 이진 탐색 트리는 왼쪽에는 작은 값을 오른쪽에는 큰값을 두는 자료구조로 검색시 $\log_2 N$의 성능을 갖는 성능의 뛰어난 검색 자료구조이다.

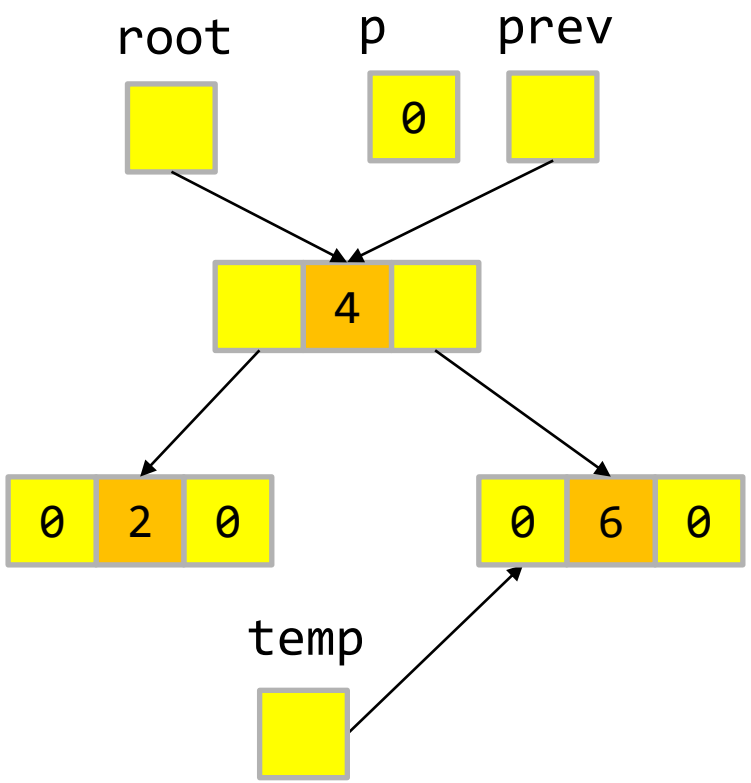


O(n)

O($\log_2 n$)

Binary Search Tree 삽입의 구현 I

```
void insert_data( int data ){
    NODE *temp, *p= root, *prev;
    temp = malloc( sizeof(NODE) );
    temp->data = data;
    temp->left = temp->right = 0;

    if( root == 0 )   {
        root = temp;
        return ;
    }
    while( p )   {
       prev = p;
       if( p->data > data )
          p = p->left
       else if ( p->data < data )
          p = p->right;
       else
          return ;
    }
    if( prev->data > data  )
        prev->left = temp;
    else
        prev->right = temp;
}
```



이진 탐색 트리 삽입의 최적화

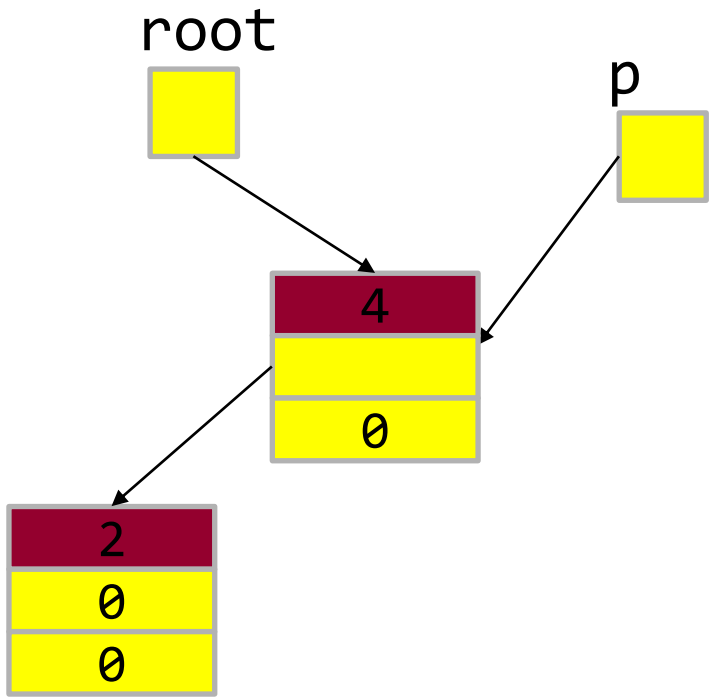
```
void insert_data( int data ){
    NODE *temp, *p= root, *prev;
    temp = malloc( sizeof(NODE) );
    temp->data = data;
    temp->left = temp->right = 0;

    if( root == 0 )   {
        root = temp;
        return ;
    }
    while( p )   {
       prev = p;
       if( p->data > data )
          p = p->left
       else if ( p->data < data )
          p = p->right;
       else
          return ;
    }
    if( prev->data > data  )
        prev->left = temp;
    else
        prev->right = temp;
}
```

문제점

1. prev가 필요한가?
2. root=0일때의 코드가 일반화 되어 있지 않다.
3. prev의 백업이 있다.
4. 루프문 후 조건 비교가 필요한가?

이중 포인터를 이용한 삽입 최적화

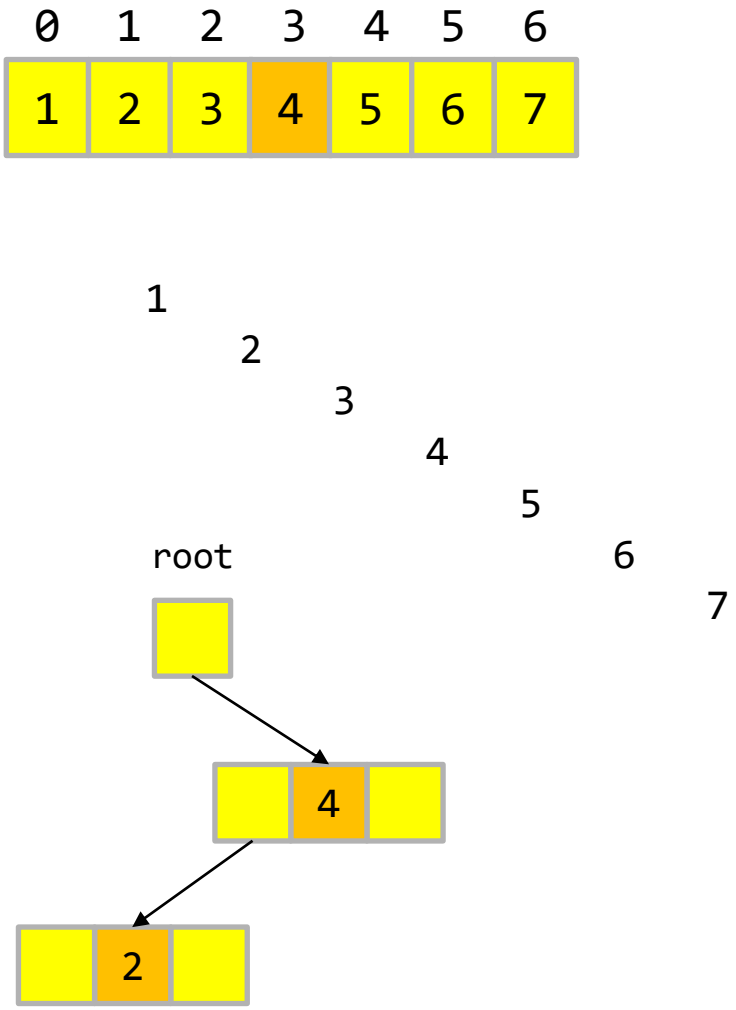
```
void insert_data( int data )
{
    NODE *temp, **p= &root;
    temp = malloc( sizeof(NODE) );
    temp->data = data;
    temp->left = temp->right = 0;

    while( *p )
    {
        if( (*p)->data > data )
            p = &(*p)->left
        else if ( (*p)->data < data )
            p = &(*p)->right;
        else
            return ;
    }
    *p = temp;
}
```



RB tree의 삽입코드의 구현 예

```
static inline struct page * __rb_insert_page_cache(
                            struct inode * inode,
                            unsigned long offset,
                            struct rb_node * node){
    struct rb_node ** p = &inode->i_rb_page_cache.rb_node;
    struct rb_node * parent = NULL;
    struct page * page;

    while (*p)    {
        parent = *p;
        page = rb_entry(parent, struct page, rb_page_cache);

        if (offset < page->offset)
            p = &(*p)->rb_left;
        else if (offset > page->offset)
            p = &(*p)->rb_right;
        else
            return page;
    }
    *p = node;
}
```

이진 탐색 트리의 밸런스 일반화




```
void __fill( NODE *temp, int *a, int *n)
{
    if( temp == 0  )
        return ;
    __fill(temp->left, a, n);
    a[(*n)++] = temp->data;
    __fill(temp->right, a, n);
}

NODE* __bal( int *a, int n)
{
   int mid = n/2;
   NODE *temp;

   temp = malloc(sizeof(NODE));
   temp->data = a[mid];
   temp->left = __bal(a,mid) ;
   temp->right = __bal(a+mid+1, n-mid-1);
   return temp;
}
```